Multicore architectures are increasingly used in embedded systems to achieve higher throughput with lower energy consumption. This trend accentuates the need to convert existing sequential code to effectively exploit the resources of these architectures. We present a parallelization flow and toolset for legacy C code that includes a performance estimation tool, a parallelization tool, and a streaming-oriented parallelization framework. These are part of the work-in-progress EU FP7 PHARAON project that aims to develop a complete set of techniques and tools to guide and assist software development for heterogeneous parallel architectures. We demonstrate the effectiveness of the use of the toolset in an experiment where we measure the parallelization quality and time for inexperienced users, and the parallelization flow and performance results for the parallelization of a practical example of a stereo vision application.

Categories and Subject Descriptors
D.1 [Programming Techniques]: Concurrent Programming—Parallel programming

General Terms
execution profiling, data dependency analysis, program parallelization, energy estimation

1. INTRODUCTION

Latest market evolution shows a significant increase of the use of multicore architectures [4]. Over the last decade, processors and systems refocus from single-thread execution acceleration to overall throughput increase on multi-processor architectures. While these were traditionally used in specific domains with very high processing needs, they gradually permeated to many embedded systems, which increased the need to parallelize massive amounts of legacy sequential code [3, 10]. However, efficient parallel programming is still challenging for new projects [11, 2] as for the legacy ones.

The revolution in hardware architectures challenges the software development techniques to efficiently exploit the potential of the multicore architectures, including the performance-power trade-offs that are often important for portable embedded systems. Automated software parallelization has been extensively explored especially at the statement, basic block and loop levels, which are appropriate for VLIW and vector processors [21, 9]. By contrast, the tools for the exploration of parallelization opportunities at task level that is best suited for modern multi-core processors, were less addressed, with some notable exceptions [14, 6]. However, most of the latter are so far restricted to specific types of loops and data access patterns.

The PHARAON project aims to enable the development of complex systems with high processing needs and low-power requirements. Figure 1 shows the techniques and tools developed to this end. The first set addresses the de-
1.1 Evolution beyond the state of the art

Although long studied, compilers can generally extract a limited level of parallelism unless they are used for special applications and, often, for specific coding styles [13, 14]. Efforts like MORPHEUS [26], CRISP [1] and MEGHA [23] are usually tailored to the target architecture they produce parallel code for. Semi- or full-automatic approaches [7, 19, 27, 28] generally offer limited means for manual analysis and improvement of the tool-generated solutions. Also, automatic code generation or modification is seldom accepted in large industrial projects due to maintainability and debugging concerns. Dominant industry players have also proposed several compilation and debugging tools. For example, OpenCL and CUDA extend the C language to generate efficient code for GPUs. In this project, OpenStream extends the OpenMP standard, better suited for CPUs, with streaming-oriented constructs.

UML is a common modeling language for high-level system design [20]. Semi-automatic generation of HW/SW infrastructures [5] and a flow to dynamically reconfigure SoCs [12] from UML have been proposed, as well as low-power run-time management and scheduling, most recent by dynamic voltage and frequency scaling [8, 25]. The design-time approaches use slow integer linear programming [24] and cannot be used at run time. PHARAON proposes a complete framework addressing heterogeneous multi-processor platforms for power consumption optimization.

1.2 System design flow

The design flow extends from high-level UML specification to target platform programming (see Figure 2). The specification can handle homogeneous, heterogeneous and distributed systems, and can be used to generate code for performance and parallelization analysis, and power management for the optimal use of the target platform resources.

The first stage uses the Pareon performance analysis tool to evaluate the timing and energy of the C code of the UML components. Code parallelization in the second stage is interactively driven by ParTools and the optimized code is either simulated again or implemented and analyzed on the target platform in the third stage. This helps assessing the parallelization quality and to extract the information for run-time optimizations. The latter are used by the reconfiguration manager and the low-power scheduler that control the physical platform at run-time to provide the required application performance with minimal energy consumption.

2. WORKFLOW FOR PARALLELIZATION

Parallelization using only a classical source code profiler is not trivial without a good knowledge of the code. For instance, a typical gprof profile shown in Listing 1 shows clearly the most computation-intensive parts of the program, but it does not provide any information on data flows and dependencies, which are well known parallelization inhibitors.

Other tools provide more details, but rarely provide a comprehensive view of the data dependencies at program level.

Listing 1: Typical execution profile (gprof) output

<table>
<thead>
<tr>
<th>time</th>
<th>sec.</th>
<th>sec.</th>
<th>calls</th>
<th>s/call</th>
<th>s/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.61</td>
<td>2.79</td>
<td>2.79</td>
<td>788215425</td>
<td>0.00</td>
<td>0.00 Dot</td>
</tr>
<tr>
<td>13.78</td>
<td>5.12</td>
<td>2.32</td>
<td>141631877</td>
<td>0.00</td>
<td>0.00 IntersectQuad</td>
</tr>
<tr>
<td>8.28</td>
<td>6.50</td>
<td>1.39</td>
<td>281277610</td>
<td>0.00</td>
<td>0.00 IntersectSphere</td>
</tr>
<tr>
<td>7.90</td>
<td>9.19</td>
<td>1.33</td>
<td>69361053</td>
<td>0.00</td>
<td>0.00 NormalizeVec3</td>
</tr>
</tbody>
</table>

For these reasons, the PHARAON workflow for parallelization collects program-wide data dependencies at run-time and presents them for analysis in an abstract and intuitive way. The toolset flow does not make any specific assumptions on developer skills, parallelization method, syntax or parallelization framework:

- run-time collection of the execution profile and data dependencies of the program;
- performance analysis for energy consumption estimates and execution histograms using either simulation or measurement of the (embedded) target system;
- intuitive interactive graphical display of execution profile, data dependencies, and performance estimations;
- manual analysis of the data and selection of the most promising parallelization opportunities and style;
- parallelization, test and debug of the parallel code, and measurement of performance enhancements;
- code refactoring to improve the parallel execution performance of the algorithms.

The steps above can be iterated as needed until satisfactory results are achieved with the effort allocated to the project. In the following sections will be presented in more detail the PHARAON toolset components that support the flow. Then, the effectiveness of the use of the toolset will be demonstrated, both in terms of simplification of the parallelization task for low skill users, as well as the acceleration obtained on a stereo vision application of practical interest.

2.1 ParTools parallelization toolset

ParTools [15, 16] is a free software project designed to support the developers of various skill levels to parallelize legacy sequential C code that can include complex control structures, pointer operations, and dynamic memory allocation. ParTools was designed to facilitate the discovery of both task and data parallelization opportunities and can be used for any parallelization technique.

The toolset flow, shown in Figure 3, is divided in four stages: I source instrumentation, II execution trace profile and data dependency collection and compaction at run-time, III execution data graphical visualization and analysis, and IV source code parallelization.

An automatic annotator instruments the sequential source in stage I for run-time data dependency collection. The data generated by the instrumentation are collected and compacted at run-time in stage II by a library, and saved in the...
They are graphically displayed in stage III as a data dependency graph (DDG), with the nodes representing program control (e.g., statements, loops, function calls) and the edges representing the data dependencies. All elements are analyzed within call stacks, since the context can influence their execution parameters. The nodes for complex program structures (e.g., loops, function calls) fold all the call stacks rooted there. These can be unfolded progressively, as needed, to discover good parallelization candidates, as we will show later. Stage IV supports manual program parallelization based on the exploration above. The source code in the IDE is connected with the graph elements in the graph viewer, which also provides several methods to temporarily hide graph sections that are not relevant for the parallelization, such as graph re-rooting to any given node.

ParTools analysis can complement automatic parallelization tools (e.g., that of Compaan Design\(^3\)) which can significantly benefit from the toolset-driven program-wide data dependency analysis. ParTools can show: where the compute-intensive procedures are; if there are any data dependencies besides those through procedure arguments; whether the procedure inputs and outputs are truly unaliased; whether the procedure inputs are truly read-only and outputs are truly write-only. Also, ParTools can import data from external analysis tools that complement its analysis capabilities, such as energy analysis and execution histograms from Pareon. These are displayed on the graph to provide the developers with a more comprehensive view on program execution and help them make better parallelization decisions.

The graphical visualization opens by abstracting all execution details under the call to main() function, as shown in Figure 4. The fold label, kept minimal but informative, shows: the fold type, its estimated execution load and energy consumption (imported from the Pareon analyzer), the source file name and line, the function name and call stack ID. The folds can be progressively unfolded to help the developer uncover data relevant for parallelization opportunities. For instance, Figure 5 shows a stage in the analysis of the stereo vision application from Figure 4. The rectangular nodes correspond to loop folds and higher color intensity corresponds to higher execution load (for nodes) or higher data transfers (for edges).

Another ParTools important feature is the data dependency view of a DDG fold node. It shows the node read and write data dependencies which are essential for any parallelization mechanism, language and style. The view is organized in layers, as shown in the excerpt in Figure 6. The top layer shows the leaf nodes (C statements) that produce the

---

\(^3\)Compaan Design BV [http://www.compaandesign.com/](http://www.compaandesign.com/)
The two loop folds (square shape) with stronger colourization include 53% and 18% of program execution. With no strong data dependencies between them, they make good parallelization candidates.

Incoming data (displayed on the next layer). The middle layer shows the nodes (C statements) in the fold that exchange data with nodes outside the fold. The bottom layer displays the nodes outside the fold that use the data produced within it. These fold-level dependencies are typically difficult to extract through static code analysis or inspection, since the producers and consumers can be at various depths and in different call stacks, and use any type of data (dynamic, local, global, etc.)

ParTools can insert these dependencies as comments in the source code and an OpenMP pragma template, which can be adapted to parallelize the fold (see Figure 7).

2.2 Performance analysis in Pareon

2.2.1 Performance analysis toolflow

The Pareon tool-suite features leading-edge analysis and interactive parallelization capabilities. In the toolset flow of
the PHARAON project it provides performance analysis of C and C++ applications on the target hardware platforms (currently ARM Cortex A9 and an Intel Core 5), including energy consumption estimation. These data are imported by ParTools to provide the developers with a comprehensive view of the run-time effects of the program that can help them make effective parallelization decisions. The energy estimates are also used by the low power scheduler to select the most power-efficient operating mode of the system. Pareon can also analyze parallel C and C++ programs that use POSIX threads or OpenMP pragmas (the latter are under test), allowing to check the effects of the parallelization decision and close the loop of the PHARAON toolset.

The internal Pareon flow for performance analysis is shown in Figure 8. It offers both command line interface (CLI) tools and a GUI. The CLI tools are used to automate the interface with the PHARAON project toolset, while the GUI allows the developers to inspect the results of the modelling. The vfcc compiler is one of the most important CLI tools that translates the source code into a generic executable for a target-independent intermediate instruction set architecture. This code is then run by the Pareon simulator using the necessary test data, input files, environment variables, etc. to collect various statistics. These can then be converted into estimates for a particular hardware target platform using report commands.

Pareon performance analysis is only a few hundred times slower than native execution, which is much faster than the usual gate-level back-annotated timing and power modelling tools in the EDA industry. Due to architecture virtualization, Pareon can model configurations that do not exist (yet)

\[^4\] The extensive Pareon documentation is available online at [http://www.vectorfabrics.com/docs/pareon/current/](http://www.vectorfabrics.com/docs/pareon/current/)

### 2.2.2 Performance histograms

Pareon performance analysis can generate at run-time timing and invocation count statistic histograms for functions and loops. These can benefit parallelization decisions since code execution may depend on the context (e.g., on function arguments). For example, the execution time of the loop body in Listing 2 depends on the function argument.

#### Listing 2: Varying function timing

```c
int foo(int n)
{
    int s = 0, i;
    for (i = 0; i < n; i++)
        s += i * i;
    return s;
}

int main()
{
    int val[] = { 5, 4, 2, 3, 4, 3, 5 };
    int i;
    for (i = 0; i < 8; i++)
        val[i] = foo(val[i]);
    return 0;
}
```

Thus, parallelizing the loop in the `main()` function using a round-robin scheduling is inefficient, since the invocation time is not constant. This would lead to imbalanced load and lower speedups than for a dynamic scheduling.

Pareon histograms can be explored using the tool GUI or can be exported for integration in other tools, e.g., to complement the call stack-based analysis of ParTools. For example, Figure 9 shows the timing histogram for a loop, including the number of times it has been executed and how much time each execution took, grouped in time bins. Clicking on an iteration bin displays its details as shown in Figure 10 for bin #10.

Histograms often exhibit specific patterns. For a spike with few iterations per invocation, the speedup of this invocation is limited by the parallelization overhead; one spike with many iterations per invocation generally benefits from parallelization; and loops with non-constant body execution time may benefit most from a dynamic scheduling to avoid workload imbalance.
2.3 OpenStream: OpenMP extension for data-flow and stream parallelism

OpenStream\(^5\) is a stream programming language, designed as an incremental extension to the OpenMP parallel programming language [22]. It allows expressing arbitrary task-level data flow dependence patterns through compiler annotations (pragmas) that dynamically generate a streaming program. The language supports nested task creation, modular composition, variable and unbounded sets of producers/consumers, and first-class streams. These features allow translating high-level parallel programming patterns into efficient data-flow code. OpenStream is provided as a tightly integrated collection of compilation, code generation, and concurrent runtime algorithms for task-level parallel programming, particularly effective on embedded multicores.

Data-flow execution is essential to reduce energy consumption, one of the primary focuses of the PHARAON project, by reducing the severity of the memory wall in two complementary ways: (1) thread-level data flow naturally hides latency and (2) decoupled producer-consumer pipelines favor on-chip communication, bypassing global memory. Furthermore, OpenStream exceeds the performance of state-of-the-art parallel programming environments like StarSs. Figure 11 shows comparatively that OpenStream speedups against sequential execution (solid) exceed those of StarSs (dashed) for a block-sparse matrix LU factorization on a dual-socket AMD Opteron Magny-Cours 6164HE machine with 2\(\times\)12 cores at 1.7 GHz.

This performance advantage is due to the optimized runtime of OpenStream, enabling very low-overhead synchronization [22]. In particular, it relies on a work-stealing scheduler that improves on Chase and Lev’s concurrent doubly-ended queue for relaxed memory models. This algorithm has been ported to x86 and ARM architectures, the latter being optimized and proven correct, leveraging on recent progress in the formalization of memory consistency [17]. Our results show that the optimized ARM code outperforms the original sequentially consistent Chase–Lev in a variety of benchmarks, including a selection of standard fine-grained task-parallel computations.

To support advanced stream-computing patterns with sliding windows and batched communications, we conducted complementary experiments with a single-producer, single-consumer (SPSC) FIFO queue. Beyond OpenStream, this concurrent data structure is essential for parallel languages and embedded multiprocessors: it arises from a variety of parallel design patterns and from the distribution of Kahn process networks over multiprocessor architectures. With WeakRB [17], we provide a portable, correct and efficient concurrent implementation of data-flow streams in C11. It uses advanced caching and batching extensions, and leverages the low-level atomics in C11 with relaxed memory consistency. We validated its portability and performance on 3 architectures with diverse hardware memory models, including 2 embedded platforms. Figure 12 shows how WeakRB outperforms one of the state of the art algorithms, MCRB [18], on an ARM multicore platform, sustaining close-to peak throughput in core-to-core streaming communications.

2.4 Tool support: interface and automation

ParTools toolset is made of several free software projects integrated under the control of the IDE [16]. The parallelization flow described in Section 2 starts with the compilation of the sequential code using the partools-
were used as baseline to assess the effects of using the toolset.

and OpenMP parallelization pragmas) and its results
only standard code analysis and development tools (such as
partitioned in two sets, of which one was required to use
algorithm, and a cascade of two FIR filters. The groups were
iment without any knowledge of writing parallel software.

Process Networks, but had never written code using this
model). They were also exposed to the concept of Kahn
engineering master (5th year overall) that covers modelling
students from a second year course for the electronics en-
allelize a previously unknown legacy application. We used
helps relatively inexperienced users to more effectively par-

The ParTools modular structure allows to easily integrate
data from external tools, for example from the Pareon anal-
alysis tool as shown in the PHARAON parallelization flow
in Section 2. The energy and timing estimations from the
Pareon toolset are displayed on the DDG to allow the de-
veloper to explore the best parallelization opportunities in
terms of both execution speed-up and energy consumption
reduction by focusing on: (1) folds that account for im-
portant parts of program execution, starting by unfolding
the main() fold (see Figure 4) and (2) on important data
flows that are highlighted in the DDG by prominent arrows.
The high execution folds can be good candidates to data-
parallelization, e.g., using OpenMP. At the same time, if
these folds are connected by important data transfers, they
may be good candidates for task-parallelization, e.g., using
OpenStream.

Moreover, the input and output data dependencies for the
folds considered for parallelization can be analyzed using the
detailed data dependency view shown in Figure 6. This view
emphasizes the direction of the dependency (read or write),
what source statements produce and consume it, where the
variables holding them were declared, and if there are hidden
data dependencies (e.g., on global data) that are not visible
at the level in the call stack considered for parallelization.

3. EXPERIMENTAL EVALUATION

3.1 Comparative use test

To illustrate the benefits of the PHARAON toolset flow
in this respect, we present the results of a comparative use
test. Its purpose is to show how the use of the toolset helps
relatively inexperienced users to more effectively par-
allelize a previously unknown legacy application. We used
students from a second year course for the electronics en-
gineering master (5th year overall) that covers modelling
languages, such as SystemC, Esterel and Kahn Process net-
works, and the associated synthesis and verification algo-
rithms and tools. The course does not teach specifically
how to parallelize software. The students had only used the
SystemC language to model multiple threads communicat-
ing via signals (i.e., using the Moore synchronous reactive
model). They were also exposed to the concept of Kahn
Process Networks, but had never written code using this
computation model. Hence, the students entered the exper-
iment without any knowledge of writing parallel software.

The test assignment was to analyze and parallelize three
real-life use cases: an MJPEG encoder, a ray tracing algo-
rithm, and a cascade of two FIR filters. The groups were
partitioned in two sets, of which one was required to use
only standard code analysis and development tools (such as
gprof and OpenMP parallelization pragmas) and its results
were used as baseline to assess the effects of using the toolset.
The other set was required to use the PHARAON toolset in
addition to the standard tools of the first set and its results
were evaluated against the results of the first set separately,
for each parallelization candidate program.

The students were requested to spend at most a couple of
days on the parallelization. Only 9 groups out of 11 com-
pleted the assignment and the results of the test are summa-
rized in Figure 13. The X axis lists the test cases as follows:
“mjpeg” is an MJPEG encoding algorithm with an acyclic
data dependency graph at the top level; “FIR” is a couple
of cascaded FIR filters; “raytracer” is a ray tracing applica-
tion with a well known top-level data parallelism. The tools
used for the parallelization are: only standard development
and analysis tools for “no toolset” and both standard and the
PHARAON toolset for “toolset”. The Y axis shows the
time (in 8-hour days) needed to complete the various phases
of the parallelization assignment, and the speedup obtained
on a 4-core Intel architecture.

The graph indicates: the training time to get acquainted
with the tools; the time to perform the first parallelization
(discover parallelism, analyze the data dependencies, write
the parallel code using OpenMP pragmas, and debug the
results so that the execution was correct); the time to further
optimize the parallelized code to improve the speedup; and
the final speedup to sequential execution.

The results of the “mjpeg” test show that using the toolset
considerably reduced the parallelization time, but at the cost
of more training time. Additionally, more time invested to
learn the toolset appears to pay off by reducing the par-
allelization time later. The final speedup results are similar,
with some variability that does not appear to depend on the
use of the toolset. The “FIR” test shows that a group using
the toolset was the only one obtaining any speedup. Learn-
ing how to use the toolset in this case took a long time. Also
for the “raytracer” test, the use of the toolset reduced the
parallelization time at the cost of more training time. The
use of the toolset lead to slightly better speedup than with-
out. However, neither group obtained a functionally correct
parallelization since they missed some of the data dependen-
cies due to the incompleteness of their code analysis. Up to
a point this is unavoidable because of the “optimistic”, trace-
based, manual parallelization approach used. However, this
prompted us to extended the toolset after the experiment
with the capability to insert the data dependencies as com-
ments in the source code as shown in Figure 7.

3.2 Stereo vision use case

Stereo vision applications infer 3D scene geometry from
two images with different viewpoints by calculating a dense
disparity or depth map from a pair of images under known
camera configuration. The algorithm that was used builds a
prior on the disparities by forming a triangulation on a set
of support points which can be robustly matched, reducing
the matching ambiguities of the remaining points. This
allows an efficient exploitation of the disparity search space,
yielding an accurate dense reconstruction without the need
for global optimization.

The flow of the PHARAON parallelization toolset de-
scribed in Section 2 was used on the application code as pre-
presented in Section 2.4. Unfolding the highest level of abstrac-
tion shown in Figure 4 revealed that the fold of function pro-
cess() call holds almost all program execution. Unfolding
this one shows that function process_disp() folds 99.85% of
program execution. Unfolding it reveals right away, by color
and the code we deduce that the two calls to `computeMatchingDisparity()` are independent and that the iterations of the outer loop do not show major unbalances. Thus, its execution can be sliced and executed in parallel using the OpenMP pragma shown on top of Listing 3.

**Listing 3: Contents of `computeMatches()` function**

```c
#pragma omp parallel for
for (u_can=1; u_can<D_can_width; u_can++) {
    ... 
    for (v_can=1; v_can<D_can_height; v_can++) {
        ... 
        d=computeMatchingDisparity(&pu,&pv, ... 
        if (d>=0) { 
            ... 
            computeMatchingDisparity(&pdif,&pv, ... 
            ... 
        }
    }
}
```

A similar analysis shows that the best parallelization for the other two fold candidates in Figure 14 (i.e., `mean()` and `computeDisparity()`) can be their calls in `process_disp()`, as shown in Listing 4 for the latter. The dependency analysis shows that the two calls are independent and can be executed in parallel, as shown in Listing 5. The call to `mean()` is analyzed and parallelized analogously.

**Listing 4: Call of `computeDisparity()` function**

```c
computeDisparity(p_support,tri_1, ... 
computeDisparity(p_support,tri_2, ... 
```

**Listing 5: Parallelization of `computeDisparity()` call**

```c
#pragma omp parallel sections
{
    #pragma omp section 
    { computeDisparity(p_support,tri_1, ... 
    } 
    #pragma omp section 
    { computeDisparity(p_support,tri_2, ... 
    } 
}
```

The speedup of these parallelizations was measured on the target architecture made of a 2-core 4-thread Intel i5 running at 2.6 GHz (i5-3230M) processing a set of images of

---

**Figure 13: Results of the test of the toolset use for parallelization.**

**Figure 14: Unfold of folds**

100% → `main()` 100% → `process()` 99.85% for the stereo vision application shows (by colorization intensity) three parallelization candidates: `computeMatches()` 53.11% (top), `computeDisparity()` 22.61% (bottom), and `mean()` 17.61% (right).

Intensity, that three folds hold most of the execution load, as can be seen in the center of Figure 14: `computeMatches()` 53.11% (top), `computeDisparity()` 22.61% (bottom), and `mean()` 17.61% (right). We also notice that the data dependencies between these are not very strong, suggesting that these folds may be suited for data-parallelism. In the corresponding source code, we find out that in fold `computeMatches()` the function `computeMatchingDisparity()` is called twice within the body of the innermost of two nested loops, as shown in Listing 3 (without the leading pragma).

Analyzing the data dependency view, the loop histogram
1024 × 768 pixels. The results reported in Table 1 show the effectiveness of the analysis using the PHARAON toolset to find good parallelization opportunities.

OpenStream parallelization followed similar patterns since no inter-task dependencies with streams looked promising. However, unlike the OpenMP-based parallelization, OpenStream focused on lower granularity parts of the code, leveraging the efficiency of its run-time. The results obtained on a 4-core 8-thread Intel i7 running at 2.2 GHz (i7-2720QM) are reported in Table 2. The finer grain parallelization enables greater scalability. Interestingly, it also favors data reuse across the different phases of the algorithm, which should in turn reduce energy consumption.

### 3.3 Energy consumption optimization

Based on the stereo vision application described above we evaluated the impact of parallelization on energy consumption. The Pareon tool has a built-in energy consumption model, which can predict energy consumption on ARM and Intel multicore processors, including L2 and L3 on-chip caches. The model has been developed within the PHARAON project.

In our experiment we focus on an Intel i5-2500, running at 3.2 GHz for performance and 1.6 GHz for low power consumption. For the experiment the hotspot loop in the ComputeMatches() function has been selected, which Pareon estimated to account for about 60% of the total run-time. Experimentation with an impact of parallelization on timing and energy consumption in Pareon can be done within the GUI without modifying the source code. First, we sampled estimated energy consumption for the same clock frequency of 3.2 GHz in both the sequential version of the loop and the parallel version with 4 POSIX threads. Then, in the Pareon GUI we lowered the clock frequency and, consequently, the supply voltage to save even more energy and recorded the energy consumption again. Noteworthy, thanks to faster parallel execution we did not compromise performance, yet reduced energy consumption. The recorded results are shown in Table 3 and correspond to the hotspot loop only, excluding other (sequential) parts of the program.

Remarkably, even without reducing the clock frequency (and the supply voltage) the parallelization can lead to energy savings, see the 3rd column with 4 threads and 3.2 GHz. This is due to the substantial power leakage of Intel processors and the background OS tasks. Indeed, thanks to parallelization the processor executes stereo vision algorithms faster, reducing this way the leakage energy consumption overhead. Note, that in the sequential version “idle” cores keep on dissipating leakage energy negatively impacting the total figure. Moreover, dropping the clock frequency and the supply voltage reduces the energy consumption down to 41% of the sequential version. Interestingly, the parallel execution was 2.6 times faster than that of the original sequential execution despite the lower clock frequency.

### 4. CONCLUSION

This work presents the toolset and techniques developed in the PHARAON project with particular emphasis on the support for parallelization of legacy C code for multiprocessors platforms. These implement a complete flow, from UML modeling to final implementation, helping to reduce the development time, to increase the performance and to reduce the energy consumption.

The parallelization flow includes several tools. A performance estimation tool (Pareon) is used to extract timing and energy estimations for the code under analysis. A parallelization tool (ParTools) performs execution profiling and collects data dependencies program-wide at run-time. These, along with performance estimations, are shown in an interactive analysis interface at selectable levels of abstraction and analysis to help the developer decide on the best parallelization techniques and opportunities. The support for streaming-oriented parallelization is provided by OpenStream, an extension to the OpenMP standard.

The effectiveness of the parallelization toolset is demonstrated on two practical cases. One is a use case, involving inexperienced users, demonstrates the increment in parallelization quality and reduction of parallelization time due to the use of the toolset. The other demonstrates the use of the toolset for the parallelization of a stereo vision application of practical interest. The toolset helps to identify good parallelization candidates, at the proper level, and analyze their data dependencies and execution timings to define the best parallelization technique to achieve a significant speedup.

### 5. ACKNOWLEDGMENTS

This work is performed in the framework of the FP7-288307 funded project PHARAON. The stereo vision application has been provided by Tedesys within the project.

### 6. REFERENCES

